Chapter 9

How to Develop LSTMs for Time Series Forecasting

Long Short-Term Memory networks, or LSTMs for short, can be applied to time series forecasting. There are many types of LSTM models that can be used for each specific type of time series forecasting problem. In this tutorial, you will discover how to develop a suite of LSTM models for a range of standard time series forecasting problems. The objective of this tutorial is to provide standalone examples of each model on each type of time series problem as a template that you can copy and adapt for your specific time series forecasting problem.

After completing this tutorial, you will know:

- How to develop LSTM models for univariate time series forecasting.

- How to develop LSTM models for multivariate time series forecasting.

- How to develop LSTM models for multi-step time series forecasting.

Let's get started.

9.1 Tutorial Overview

In this tutorial, we will explore how to develop a suite of different types of LSTM models for time series forecasting. The models are demonstrated on small contrived time series problems intended to give the flavor of the type of time series problem being addressed. The chosen configuration of the models is arbitrary and not optimized for each problem; that was not the goal. This tutorial is divided into four parts; they are:

1. Univariate LSTM Models

2. Multivariate LSTM Models

3. Multi-step LSTM Models

4. Multivariate Multi-step LSTM Models

9.2 Univariate LSTM Models

LSTMs can be used to model univariate time series forecasting problems. These are problems comprised of a single series of observations and a model is required to learn from the series of past observations to predict the next value in the sequence. We will demonstrate a number of variations of the LSTM model for univariate time series forecasting. This section is divided into six parts; they are:

1. Data Preparation

2. Vanilla LSTM

3. Stacked LSTM

4. Bidirectional LSTM

5. CNN-LSTM

6. ConvLSTM

Each of these models are demonstrated for one-step univariate time series forecasting, but can easily be adapted and used as the input part of a model for other types of time series forecasting problems.

9.2.1 Data Preparation

Before a univariate series can be modeled, it must be prepared. The LSTM model will learn a function that maps a sequence of past observations as input to an output observation. As such, the sequence of observations must be transformed into multiple examples from which the LSTM can learn. Consider a given univariate sequence:

```
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Listing 9.1: Example of a univariate time series.

We can divide the sequence into multiple input/output patterns called samples, where three time steps are used as input and one time step is used as output for the one-step prediction that is being learned.

```
X,              y
10, 20, 30,     40
20, 30, 40,     50
30, 40, 50,     60
...
```

Listing 9.2: Example of a univariate time series as a supervised learning problem.

The `split_sequence()` function below implements this behavior and will split a given univariate sequence into multiple samples where each sample has a specified number of time steps and the output is a single time step.

```
# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 9.3: Example of a function to split a univariate series into a supervised learning problem.

We can demonstrate this function on our small contrived dataset above. The complete example is listed below.

```
# univariate data preparation
from numpy import array

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 9.4: Example of transforming a univariate time series into a supervised learning problem.

Running the example splits the univariate series into six samples where each sample has three input time steps and one output time step.

```
[10 20 30] 40
[20 30 40] 50
[30 40 50] 60
```

```
[40 50 60] 70
[50 60 70] 80
[60 70 80] 90
```

Listing 9.5: Example output from transforming a univariate time series into a supervised learning problem.

Now that we know how to prepare a univariate series for modeling, let's look at developing LSTM models that can learn the mapping of inputs to outputs, starting with a Vanilla LSTM.

9.2.2   Vanilla LSTM

A Vanilla LSTM is an LSTM model that has a single hidden layer of LSTM units, and an output layer used to make a prediction. Key to LSTMs is that they offer native support for sequences. Unlike a CNN that reads across the entire input vector, the LSTM model reads one time step of the sequence at a time and builds up an internal state representation that can be used as a learned context for making a prediction. We can define a Vanilla LSTM for univariate time series forecasting as follows.

```python
# define model
model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(n_steps, n_features)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

Listing 9.6: Example of defining a Vanilla LSTM model.

Key in the definition is the shape of the input; that is what the model expects as input for each sample in terms of the number of time steps and the number of features. We are working with a univariate series, so the number of features is one, for one variable. The number of time steps as input is the number we chose when preparing our dataset as an argument to the `split_sequence()` function.

The shape of the input for each sample is specified in the `input_shape` argument on the definition of first hidden layer. We almost always have multiple samples, therefore, the model will expect the input component of training data to have the dimensions or shape: `[samples, timesteps, features]`. Our `split_sequence()` function in the previous section outputs the X with the shape `[samples, timesteps]`, so we easily reshape it to have an additional dimension for the one feature.

```python
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
```

Listing 9.7: Example of reshaping training data for the LSTM.

In this case, we define a model with 50 LSTM units in the hidden layer and an output layer that predicts a single numerical value. The model is fit using the efficient Adam version of stochastic gradient descent and optimized using the mean squared error, or 'mse' loss function. Once the model is defined, we can fit it on the training dataset.

```python
# fit model
model.fit(X, y, epochs=200, verbose=0)
```

Listing 9.8: Example of fitting the LSTM model.

After the model is fit, we can use it to make a prediction. We can predict the next value in the sequence by providing the input: `[70, 80, 90]`. And expecting the model to predict something like: `[100]`. The model expects the input shape to be three-dimensional with `[samples, timesteps, features]`, therefore, we must reshape the single input sample before making the prediction.

```python
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
```

Listing 9.9: Example of preparing an input sample ready for making an out-of-sample forecast.

We can tie all of this together and demonstrate how to develop a Vanilla LSTM for univariate time series forecasting and make a single prediction.

```python
# univariate lstm example
from numpy import array
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
# define model
model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(n_steps, n_features)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=200, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps, n_features))
```

```
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.10: Example of a Vanilla LSTM for univariate time series forecasting.

Running the example prepares the data, fits the model, and makes a prediction. We can see that the model predicts the next value in the sequence.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[102.09213]]
```

Listing 9.11: Example output from a Vanilla LSTM for univariate time series forecasting.

9.2.3 Stacked LSTM

Multiple hidden LSTM layers can be stacked one on top of another in what is referred to as a Stacked LSTM model. An LSTM layer requires a three-dimensional input and LSTMs by default will produce a two-dimensional output as an interpretation from the end of the sequence. We can address this by having the LSTM output a value for each time step in the input data by setting the `return_sequences=True` argument on the layer. This allows us to have 3D output from hidden LSTM layer as input to the next. We can therefore define a Stacked LSTM as follows.

```
# define model
model = Sequential()
model.add(LSTM(50, activation='relu', return_sequences=True, input_shape=(n_steps,
    n_features)))
model.add(LSTM(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

Listing 9.12: Example of defining a Stacked LSTM model.

We can tie this together; the complete code example is listed below.

```
# univariate stacked lstm example
from numpy import array
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense

# split a univariate sequence
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
```

```
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
# define model
model = Sequential()
model.add(LSTM(50, activation='relu', return_sequences=True, input_shape=(n_steps,
    n_features)))
model.add(LSTM(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=200, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.13: Example of a Stacked LSTM for univariate time series forecasting.

Running the example predicts the next value in the sequence, which we expect would be 100.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[102.47341]]
```

Listing 9.14: Example output from a Stacked LSTM for univariate time series forecasting.

9.2.4   Bidirectional LSTM

On some sequence prediction problems, it can be beneficial to allow the LSTM model to learn the input sequence both forward and backwards and concatenate both interpretations. This is called a Bidirectional LSTM. We can implement a Bidirectional LSTM for univariate time series forecasting by wrapping the first hidden layer in a wrapper layer called Bidirectional. An example of defining a Bidirectional LSTM to read input both forward and backward is as follows.

```
# define model
model = Sequential()
model.add(Bidirectional(LSTM(50, activation='relu'), input_shape=(n_steps, n_features)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

Listing 9.15: Example of defining a Bidirectional LSTM model.

The complete example of the Bidirectional LSTM for univariate time series forecasting is listed below.

```python
# univariate bidirectional lstm example
from numpy import array
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import Bidirectional

# split a univariate sequence
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
# define model
model = Sequential()
model.add(Bidirectional(LSTM(50, activation='relu'), input_shape=(n_steps, n_features)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=200, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.16: Example of a Bidirectional LSTM for univariate time series forecasting.

Running the example predicts the next value in the sequence, which we expect would be 100.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[101.48093]]
```

Listing 9.17: Example output from a Bidirectional LSTM for univariate time series forecasting.

9.2.5 CNN-LSTM

A convolutional neural network, or CNN for short, is a type of neural network developed for working with two-dimensional image data. The CNN can be very effective at automatically extracting and learning features from one-dimensional sequence data such as univariate time series data. A CNN model can be used in a hybrid model with an LSTM backend where the CNN is used to interpret subsequences of input that together are provided as a sequence to an LSTM model to interpret. This hybrid model is called a CNN-LSTM.

The first step is to split the input sequences into subsequences that can be processed by the CNN model. For example, we can first split our univariate time series data into input/output samples with four steps as input and one as output. Each sample can then be split into two sub-samples, each with two time steps. The CNN can interpret each subsequence of two time steps and provide a time series of interpretations of the subsequences to the LSTM model to process as input. We can parameterize this and define the number of subsequences as `n_seq` and the number of time steps per subsequence as `n_steps`. The input data can then be reshaped to have the required structure: `[samples, subsequences, timesteps, features]`. For example:

```
# choose a number of time steps
n_steps = 4
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, subsequences, timesteps, features]
n_features = 1
n_seq = 2
n_steps = 2
X = X.reshape((X.shape[0], n_seq, n_steps, n_features))
```

Listing 9.18: Example of reshaping data for a CNN-LSTM model.

We want to reuse the same CNN model when reading in each sub-sequence of data separately. This can be achieved by wrapping the entire CNN model in a `TimeDistributed` wrapper that will apply the entire model once per input, in this case, once per input subsequence. The CNN model first has a convolutional layer for reading across the subsequence that requires a number of filters and a kernel size to be specified. The number of filters is the number of reads or interpretations of the input sequence. The kernel size is the number of time steps included of each *read* operation of the input sequence. The convolution layer is followed by a max pooling layer that distills the filter maps down to $\frac{1}{4}$ of their size that includes the most salient features. These structures are then flattened down to a single one-dimensional vector to be used as a single input time step to the LSTM layer.

```
# define the input cnn model
model.add(TimeDistributed(Conv1D(64, 1, activation='relu'), input_shape=(None, n_steps,
    n_features)))
model.add(TimeDistributed(MaxPooling1D()))
model.add(TimeDistributed(Flatten()))
```

Listing 9.19: Example of defining the CNN input model.

Next, we can define the LSTM part of the model that interprets the CNN model's read of the input sequence and makes a prediction.

```
# define the output model
```

```
model.add(LSTM(50, activation='relu'))
model.add(Dense(1))
```

Listing 9.20: Example of defining the LSTM output model.

We can tie all of this together; the complete example of a CNN-LSTM model for univariate time series forecasting is listed below.

```python
# univariate cnn lstm example
from numpy import array
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import TimeDistributed
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 4
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, subsequences, timesteps, features]
n_features = 1
n_seq = 2
n_steps = 2
X = X.reshape((X.shape[0], n_seq, n_steps, n_features))
# define model
model = Sequential()
model.add(TimeDistributed(Conv1D(64, 1, activation='relu'), input_shape=(None, n_steps,
    n_features)))
model.add(TimeDistributed(MaxPooling1D()))
model.add(TimeDistributed(Flatten()))
model.add(LSTM(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=500, verbose=0)
# demonstrate prediction
x_input = array([60, 70, 80, 90])
```

```
x_input = x_input.reshape((1, n_seq, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.21: Example of a CNN-LSTM for univariate time series forecasting.

Running the example predicts the next value in the sequence, which we expect would be 100.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[101.69263]]
```

Listing 9.22: Example output from a CNN-LSTM for univariate time series forecasting.

9.2.6 ConvLSTM

A type of LSTM related to the CNN-LSTM is the ConvLSTM, where the convolutional reading of input is built directly into each LSTM unit. The ConvLSTM was developed for reading two-dimensional spatial-temporal data, but can be adapted for use with univariate time series forecasting. The layer expects input as a sequence of two-dimensional images, therefore the shape of input data must be: `[samples, timesteps, rows, columns, features]`.

For our purposes, we can split each sample into subsequences where timesteps will become the number of subsequences, or `n_seq`, and columns will be the number of time steps for each subsequence, or `n_steps`. The number of rows is fixed at 1 as we are working with one-dimensional data. We can now reshape the prepared samples into the required structure.

```
# choose a number of time steps
n_steps = 4
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, timesteps, rows, columns, features]
n_features = 1
n_seq = 2
n_steps = 2
X = X.reshape((X.shape[0], n_seq, 1, n_steps, n_features))
```

Listing 9.23: Example of reshaping data for a ConvLSTM model.

We can define the ConvLSTM as a single layer in terms of the number of filters and a two-dimensional kernel size in terms of (`rows, columns`). As we are working with a one-dimensional series, the number of rows is always fixed to 1 in the kernel. The output of the model must then be flattened before it can be interpreted and a prediction made.

```
# define the input cnnlstm model
model.add(ConvLSTM2D(64, (1,2), activation='relu', input_shape=(n_seq, 1, n_steps,
    n_features)))
model.add(Flatten())
```

Listing 9.24: Example of defining the ConvLSTM input model.

The complete example of a ConvLSTM for one-step univariate time series forecasting is listed below.

```
# univariate convlstm example
from numpy import array
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import ConvLSTM2D

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 4
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, timesteps, rows, columns, features]
n_features = 1
n_seq = 2
n_steps = 2
X = X.reshape((X.shape[0], n_seq, 1, n_steps, n_features))
# define model
model = Sequential()
model.add(ConvLSTM2D(64, (1,2), activation='relu', input_shape=(n_seq, 1, n_steps,
    n_features)))
model.add(Flatten())
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=500, verbose=0)
# demonstrate prediction
x_input = array([60, 70, 80, 90])
x_input = x_input.reshape((1, n_seq, 1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.25: Example of a ConvLSTM for univariate time series forecasting.

Running the example predicts the next value in the sequence, which we expect would be 100.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[103.68166]]
```

Listing 9.26: Example output from a ConvLSTM for univariate time series forecasting.

For an example of an LSTM applied to a real-world univariate time series forecasting problem see Chapter 14. For an example of grid searching LSTM hyperparameters on a univariate time series forecasting problem, see Chapter 15. Now that we have looked at LSTM models for univariate data, let's turn our attention to multivariate data.

9.3 Multivariate LSTM Models

Multivariate time series data means data where there is more than one observation for each time step. There are two main models that we may require with multivariate time series data; they are:

1. Multiple Input Series.

2. Multiple Parallel Series.

Let's take a look at each in turn.

9.3.1 Multiple Input Series

A problem may have two or more parallel input time series and an output time series that is dependent on the input time series. The input time series are parallel because each series has an observation at the same time steps. We can demonstrate this with a simple example of two parallel input time series where the output series is the simple addition of the input series.

```
# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
```

Listing 9.27: Example of defining multiple input and a dependent time series.

We can reshape these three arrays of data as a single dataset where each row is a time step, and each column is a separate time series. This is a standard way of storing parallel time series in a CSV file.

```
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
```

Listing 9.28: Example of reshaping the parallel series into the columns of a dataset.

The complete example is listed below.